VI.02.

Большинство чисто технических моментов, связанных с интерфейсами, мы разобрали. Но остались ещё несколько принципиальных вопросов: пустой интерфейс (да, его можно считать принципиальным с некоторыми оговорками, но уж очень он вездесущ, так что, если не суть, то роль его уж точно принципиальная), множественные интерфейсы (типы, которые реализуют несколько интерфейсов), встраивание (embedding) интерфейсов, ещё что-то может быть, но давайте на пару-тройку занятий остановимся, постараемся понять содержательную часть понятия интерфейс, а заодно поговорим о вечном: о сортировках и структурах данных, связанных с какими-то упорядочиваниями элементов. По ходу пьесы плавненько въедем в то, как же и где применять/использовать интерфейсы, а затем и займёмся поплотнее "ООР" в Go, точнее не-ООР (или post-ООР) в Go.

Так что в этот раз в основном повторение. На примерах сортированных структур и сортировок как раз очень хорошо показываются интерфейсы. А повторить алгоритмическую часть - это святое.

Содержание занятия получается такое:

- пару слов про сортированную коллекцию, чисто напомнить для разогрева.
- а дальше поговорим не о сортированных структурах, а о сортируемых. Грубо говоря о слайсах/массивах с возможностью сортировки. Т.е. элементы могут стоять в произвольном порядке, но у нас есть возможность их переставить - отсортировать. Так что здесь мы повторим какие-то алгоритмы сортировки: MergeSort, какие-то варианты QuickSort.
- и ещё - связные структуры; много не получится, но связный сортированный список рассмотрим. Т.е. это то же самое, что и сортированная коллекция, только реализуется это как список, т.е. поговорим о сортированном списке. Пример полезный с точки зрения встряхнуть/повторить техническую часть (связный список, работа с пойнтерами и т.д.) - на неё и напираем. А ещё это хорошее вступление к дальнейшему. А хочется дальше повторить про бинарное дерево поиска и развить его до АВЛ-дерева, всё-таки в пятом семестре места для него не хватает, а вещь хорошая.

Начнём с того, что вспомним нашу базу элементов всех наших сегодняшних структур -

интерфейс Ordered

```
package order
```

```
type Ordered interface {
    Before(b Ordered) bool
    Show()
}
```

Но с некоторыми изменениями, конечно же.

Первое - это добавили в него метод `Show()`. Это как раз совсем не очень хорошо, это неправильно так делать, но разговор о композиции интерфейсов ещё будет, так что пока оставим так. Метод вывода элементов нам понадобится, так включим его в интерфейс, а потом, не в этот раз, поговорим об этом.

Второе, гораздо более существенное - это оформим наш интерфейс в виде пакета (package). При этом, раз уж мы будем строить несколько структур - сортированных, сортируемых, да ещё и бинарную кучу, а все они используют Ordered'ы, то все они будут обращаться к этому пакету. Мало того, инерфейс Ordered имеет ценность и сам по себе, так что выделим его отдельный package. Поступим так, как авторы Go поступили с пакетом io – в нём собраны наиболее полезные интерфейсы, константы и переменные, которые так или иначе относятся к вводу-выводу. А для того чтобы не вносить дополнительных зависимостей при импорте этого пакета, есть отдельный пакет, где собраны удобные функции для работы с интерфейсами из io – пакет ioutil (io/ioutil). В итоге наш набор пакетов уляжется аналогично таким образом:

```
order
|       order.go
├──sortable
|        sortable.go
└──sorted
         sorted.go
```

Да, и всё это мы поместим в `GOROOT`, точнее в `"$GOROOT$\src"` (в моём случае это `"C:\Program Files\Go\src"`, хотя это и совершенно несущественно). Замечу, к слову, что Go 1.17 не использует больше для этих целей `GOPATH`.

Назначение пакетов.

order.go

Собственно, в нём только и есть интерфейс Ordered, и больше ничего.

sorted.go

Этот пакет мы уже начали писать на прошлом занятии, в него входят структуры, которые поддерживают упорядочение “автоматом” - элементы сразу уже при вставке становятся на положенное им место, т.е. речь идёт о структурах, которые постоянно поддерживают отсортированность своих данных. В этот раз продолжим сюжет. В него к сортированой коллекции (которая слайс) добавлен сортированный (связный , точнее, односвязный, однонаправленный) список. Правда в методе вставки в слайс поиск места для вставки нового элемента мы ищем половинным делением (на прошлом занятии это было задание для детей). Понятно, что при вставке в список никаких половинных делений нет, да и не может быть.

В пакете всё довольно просто и прозрачно, ничего особо нового (кроме интерфейса `Ordered` , конечно) нет, особых вопросов быть не должно. хорошее вступление для повторения работы со связными структурами. Единственное, что методы вставки и печати для списка написаны рекурсивно, хотя в данном случае рекурсия совсем ни к чему. Так что вот и задание для самостоятельной работы - избавиться в этих методах от рекурсии.

Пусть здесь будет текст пакета, чисто для полноты картины:
sorted.go

```
package sorted

import (
    "order"
)

type (
    SortedCollection []order.Ordered
)

func (c *SortedCollection) Insert(x order.Ordered) {
    left, right := 0, len(*c)-1
    // the insertion location is searched using the binary search method.
    // invariant: (*c)[left].Before(x) && !(*c).[right].Before(x),
    // roughly speaking: (*c)[left] < x <= (*c)[right]
    (*c) = append((*c), x)
    if right < 0 || (*c)[right].Before(x) { //right < 0 <==> empty collection
        return
    }
    if !(*c)[0].Before(x) {
```

```
        copy((*c)[1:], (*c))
        (*c)[0] = x
        return
    }
    var center int
    for right-left > 1 {
        center = (left + right) / 2
        if (*c)[center].Before(x) {
            left = center
        } else {
            right = center
        }
    }
    copy((*c)[right+1:], (*c)[right:])
    (*c)[right] = x
}

func (c SortedCollection) Print() {
    for _, x := range c {
        x.Show()
    }
}

type (
    Node struct {
        Key  order.Ordered
        Tail SortedList
    }
    SortedList struct {
        First *Node
    }
)

func NewSortedList() SortedList {
    return SortedList{} // <==> return SortedList{First:nil}
}

func (s SortedList) Empty() bool {
    return s.First == nil
}
```

```
func (s *SortedList) Insert(x order.Ordered) {
    if (*s).Empty() {
        (*s).First = &Node{Key: x, Tail: NewSortedList()}
        return
    }
    if !(*(*s).First).Key.Before(x) {
        *s = SortedList{&Node{Key: x, Tail: *s}}
        return
    }
    (*s.First).Tail.Insert(x)
}

func (s SortedList) Print() {
    if s.Empty() {
        return
    }
    (*s.First).Key.Show()
    (*s.First).Tail.Print()
}
```

sortable.go

Сортируемые структуры - коллекция (на слайсе) и связный список `Ordered`'ов. Разумеется тут втыркнуты всякие методы сортировки - и QuickSort, и MergeSort, и те и другие с вариантами. Чисто для повторения, конечно. Но интересно заметить, как мы с лёгкостью обобщили сортировки чисел из 5-го семестра на сортировки произвольных `Ordered`'ов. Да, для каждого из трёх упомянутых типов сортировки приводится только один вариант, а вариантов много, мы помним, да и в материалах к занятию многие из них они перечисляются. Ну, получаются чудные упражнения для самостоятельной работы. Впрочем, ниже они будут перечислены более или менее явно.

Sortable Collection:

- QuickSort. В пакете приводится один вариант. Разумеется, это вариант, показавший при бенчмаркинге лучшие результаты на случайном массиве - вариант Сэджвика с разделением массива, двигаясь с двух сторон. Подробно о вариантах быстрых сортировок, программы и результаты бенчмаркинга можно посмотреть в материалах к 14-му занятию пятого семестра. Прямо здесь выложены описания разных вариантов QuickSort'а в md-формате и pdf-формате. Кроме того, выложен фрагмент из "Жемчужин программирования" Бентли, посвящённый QuickSort'у

Можно, конечно, в качестве задания дать переписать другие варианты QuickSort'а на SortableCollection, но это не особо интересно - изменений там практически никаких нет. Лучше будет поиграться в другие варианты с SortableList.

- MergeSort. А вот тут в пакете представлены два варианта: рекурсивная версия сортировки бинарным слиянием (делим массив на две примерно равные части, сортируем их рекурсивно, а потом запускаем процедуру слияния полученных отсортированных половинок) и сортировку естественным слиянием (разделяем массив на возрастающие отрезки, а потом проходим неоднократно по массиву этих отрезков, сливая их попарно, до тех пор, пока не останется один отрезок - весь массив).

И тоже можно поиграться с другими вариантами сортировки слиянием (например, реализовать нерекурсивную версию сортировки слиянием), но тут тоже ничего особо интересного нет, разве что чисто для повторения.
Вот часть кода пакета `sortable`, посвящённая сортированным коллекциям:

```
type (
    SortableCollection []order.Ordered
)

func NewSortableCollection() SortableCollection {
    return []order.Ordered{}
}

const cutoff = 16

func (a SortableCollection) qSort() {
    if len(a) <= cutoff {
        return
    }

    k := rand.Intn(len(a))
    a[0], a[k] = a[k], a[0]

    pivot := a[0]
    small, large := 1, len(a)-1
    for {
        for small < len(a) && a[small].Before(pivot) {
            small++
        }
        for pivot.Before(a[large]) {
```

```
            large--
        }
        if small >= large {
            break
        }
        a[small], a[large] = a[large], a[small]
        small++
        large--
    }
    a[0], a[large] = a[large], a[0]
    a[:large].qSort()
    a[large+1:].qSort()
}

func (a SortableCollection) QSort() {
    a.qSort()
    for i, x := range a {
        j := i
        for j > 0 && x.Before(a[j-1]) {
            a[j] = a[j-1]
            j--
        }
        a[j] = x
    }
}

func (a SortableCollection) merge(start2 int) {
    res := make([]order.Ordered, len(a))
    i1, i2, ires := 0, start2, 0
    for i1 < start2 && i2 < len(a) {
        if a[i1].Before(a[i2]) {
            res[ires] = a[i1]
            i1++
        } else {
            res[ires] = a[i2]
            i2++
        }
        ires++
    }
    if i2 == len(a) {
```

```
        copy(a[ires:], a[i1:])
    }
    copy(a, res[:ires])
}

func (a SortableCollection) BinaryMergeSort() {
    if len(a) <= 1 {
        return
    }
    a[:len(a)/2].BinaryMergeSort()
    a[len(a)/2:].BinaryMergeSort()
    a.merge(len(a) / 2)
}

func (a SortableCollection) NaturalMergeSort() {
    // инициализация - заполняем слайс start
    start := []int{0}
    for i := 1; i < len(a); i++ {
        if a[i].Before(a[i-1]) {
            start = append(start, i)
        }
    }
    start = append(start, len(a))
    // сортировка
    for len(start) > 2 {
        // проходим по всему массиву, склеивая пары соседних серий
        for k := 0; k < len(start)-2; k += 2 {
            a[start[k]:start[k+2]].merge(start[k+1] - start[k])
        }
        // преобразуем слайс start: start[2] -> start[1],
        // start[4] -> start[2], start[6] -> start[3] и т.д.
        k := 0
        for {
            k += 2
            if k >= len(start) {
                break
            }
            start[k/2] = start[k]
        }
        start = start[:k/2]
```

```
        // если перед этим было нечётное количество серий, то
        // надо добавить конец последней серии - len(l)
        if start[len(start)-1] < len(a) {
            start = append(start, len(a))
        }
    }
}

func (a SortableCollection) Print() {
    for _, x := range a {
        x.Show()
    }
}
```

Sortable List:

А вот тут уже интереснее, тем более, что в пятом семестре мы этого не делали - все сортировки применяли только к слайсам/массивам. Сортируемый список мы реализуем здесь как связный однонаправленный список. Так что, конечно ни о каких движениях с двух сторон (в QuickSort'е) речи быть не может. Тем более интересно посмотреть, как это будет. И первым делом определим необходимые типы данных:

```
type (
    Node struct {
        Value  order.Ordered
        Next SortableList
    }
    SortableList struct {
        First *Node
    }
)

func (a Node) Before(b order.Ordered) bool {
    return a.Value.Before(b.(Node).Value)
}

func (a Node) Show() {
    a.Value.Show()
}
```

И сразу реализуем для типа узла списка `Node` методы Before и Show. Таким образом тип `Node` реализует интерфейс `Ordered`, так что мы сможем сравнивать (и выводить) узлы. Немного раком получается, но в некотором смысле довольно естественно - мы строим сортируемый список, он состоит из узлов, значит их надо упорядочивать. Так что сравнивать узлы можно считать нормальным, хоть и с некоторой натяжкой. И, поскольку, в пакете реализуется метод печати списка, который состоит из узлов, то и выводить узел тоже как-то надо. Понятно, что это несколько притянуто, но зато это ещё один пример реализации интерфейса. Во всяком случае код получается, пусть и не очень сильно, но прозрачнее.

Сразу приведём несколько функций общего назначения

```
func NewSortableList() SortableList {
    return SortableList{} // <==> SortableList{nil} - empty list
}

func (b SortableList) Empty() bool {
    return b.First == nil
}

func (b *SortableList) Add(x Node) {
    x.Next = *b
    (*b).First = &x
}
```

И дальше собственно сортировки.

- QuickSort.

Приведённый код реализует самый обычный вариант QuickSort'а: в качестве пивота выбираем начальный элемент списка, по оставшемуся списку пробегаем, разделяя его на два списка: те элементы, которые должны стоять перед пивотом, и те, которые должны стоять после него - это делает функция `splitByPivot`. Пробегаем, естественно, вперёд, поскольку список-то у нас однонаправленный. А затем приклеиваем пивот в конец первого списка, а вслед за ним весь второй список. На что хотелось бы обратить внимание. Новых узлов не создаётся, память под них не перераспределяется. И второе - внутри узлов значения не изменяются, никаких присваиваний значений не происходит, вся сортировка происходит только за счёт изменения связей. Это совершенно естественно, ведь сортировка - это перестановка элементов списка. И этот момент очень важен, хотя зачастую в сети встречаются материалы, в которых им пренебрегают. Это очень опасная политика - да, элементы списка мы объединили с какой-то целью, с какой-то мыслью, да, они образуют

какую-то общность, но ведь их присутствие в этом объединении не отметает их других возможных свойств и сущностей. Вполне себе возможно, что они участвуют в других каких-то сюжетах, т.е. элемент списка может быть удерживаемым (на него держит указатель) чем-то ещё. И подмена элемента недопустима. Ну, совсем тупой пример: если нам надо отсортировать класс по алфавиту, то мы переставляем фамилии в списке учеников, а не переименовываем самих учеников, а то их родители могут очень удивиться.

```
func (a *SortableList) QSort() {
    if (*a).Empty() || (*(*a).First).Next.Empty() {
        return
    }
    pivotNode := *(*a).First // var pivotNode Node
    a1, a2 := (*(*a).First).Next.separateByPivot(pivotNode)
    a1.QSort()
    a2.QSort()
    pivotNode.Next = a2
    a2.First = &pivotNode
    var runner SortableList
    if a1.Empty() {
        *a = a2
    } else {
        runner = a1
        for !(*runner.First).Next.Empty() {
            runner = (*runner.First).Next
        }
        (*runner.First).Next = a2
        *a = a1
    }
}

func (a SortableList) separateByPivot(pivot order.Ordered) (SortableList, SortableL
ist) {
    runner := a // var runner SortableList
    var next SortableList
    a1, a2 := NewSortableList(), NewSortableList()
    for !runner.Empty() {
        next = (*runner.First).Next // <==>  next = (*(runner.First)).Next
        if (*runner.First).Before(pivot) {
            (*runner.First).Next = a1
```

```
            a1 = runner
        } else {
            (*runner.First).Next = a2
            a2 = runner
        }
        runner = next
    }
    return a1, a2
}
```

Код довольно прозрачный, комментировать его, кажется, не особо нужно.

- MergeSort.
  Тоже всё довольно стандартно: делим список на два (почти) равных по размеру списка - и это самое сложное в процедуре, сортируем эти два списка рекурсивно и затем склеиваем их в один - функция `merge`. Ещё раз обратим внимание на то, что все операции с узлами списка не связаны с изменением их содержимого или с перемещением их в памяти; все операции связаны только с изменением связей между узлами, с изменением порядка их следования. Склейка носит довольнот однозначный характер, трудно здесь что-нибудь сфантазировать. А вот разделение списка на равные части - это то место, где можно пофантазировать. В нижеприведённом коде это исполняется так: заводим два бегунка, которые бегут по списку - первый на каждом шаге перемещается на один элемент, второй - на два; в итоге, когда второй бегунок добежит до конца списка, первый окажется как раз на середине. А вот, например, следующий вариант хорошо бы дать для самостоятельной работы: в первый список включаем первый, третий, пятый и т.д. элементы, а во второй, соответственно, второй, четвёртый, шестой и т.д. Для этого достаточно связать каждый узел не со следующим элементом, а с запоследующим. Кажется, что это сделать совсем несложно, но аккуратность, конечно нужна, особенно на концовке списка.

```
func (a *SortableList) MergeSort() {
    if (*a).Empty() || (*(*a).First).Next.Empty() {
        return
    }
    a1, a2 := *a, (*a.First).Next
    for !a2.Empty() && !(*a2.First).Next.Empty() {
        a1 = (*a1.First).Next
        a2 = (*(*a2.First).Next.First).Next
    }
    a2, (*a1.First).Next, a1 = (*a1.First).Next, SortableList{nil}, *a
    a1.MergeSort()
```

```
    a2.MergeSort()
    *a = merge(a1, a2)
}

func merge(a SortableList, b SortableList) SortableList {
    if a.Empty() {
        return b
    }
    if b.Empty() {
        return a
    }
    var res SortableList
    if (*a.First).Value.Before((*b.First).Value) {
        res, a = a, (*a.First).Next
    } else {
        res, b = b, (*b.First).Next
    }
    runner := res.First
    for !a.Empty() && !b.Empty() {
        if (*a.First).Value.Before((*b.First).Value) {
            (*runner).Next, a = a, (*a.First).Next
        } else {
            (*runner).Next, b = b, (*b.First).Next
        }
        runner = (*runner).Next.First
    }
    if a.Empty() {
        (*runner).Next = b
    } else {
        // b.Empty()
        (*runner).Next = a
    }
    return res
}
```

Заключительные замечания

Всё срослось. Красиво? Ну, минимум довольно естественно получилось. И интерфейсы как инструмент реализации полиморфиза (можно уже это слово

употреблять смело) работает здесь прекрасно. Но! Есть но. Всё это изрядно противоречит стратегии, можно даже сказать, идеологии Go, с его подчёркнутым отказом от наследования (и, естественно, от идеи объекта и ООП вообще). Как такие вопросы решать присущим именно Go, идиоматичным ему образом мы начнём обсуждать очень скоро, не на следующем занятии - следующее занятие проведём ещё в такой же манере, наберём чуть побольше содержательного материала, поговорим о сбалансиррованных деревьях поиска, а именно, тщательно разберёмся с АВЛ-деревьями. А вот дальше у нас появятся проблемы, которые в Go надо решать как-то иначе. И вот тут-то мы и посмотрим, что же именно предлагается для решения проблем, которые в ООП решаются с помощью наследования, а в Go совсем иначе. Так что в этом смысле Go не есть просто необъектный язык, Go - это пост-объектный язык программирования. И этот совсем иной подход дейтвительно выглядит очень здраво и естественно, в конечном итоге позволяя очень хорошо и довольно просто структурировать логику конструирования данных и методов их обработки. Какова именно эиа выигрышная стратегия в Go - это и есть главный вопрос этого семестра, и именно им мы займёмся уже совсем скоро.

Ну, и совсем уж напоследок, сведём вместе все разбросанные по тексту

задания для самостоятельной работы.

SortedList:

- Избавиться от рекурсии в методах Insert и Print.

SortableCollection:

- Написать нерекурсивный вариант сортировки бинарным слиянием.
  Вкратце суть метода такова: сначала весь массив состоит из отрезков длины *1*, понятно, что все они отсортированы *:) ;* затем идем по массиву и склеиваем первый отрезок со вторым, третий с четвёртым , пятый с шестым и т.д., если последнему отрезку не хватит пары, то так его и оставим. Получаем отрезки длины *2*. Повторяем проход, склеивая соседние отрезки - получаем отсортированные отрезки длины *4*, и т.д., пока весь массив не станет одним отсортированным отрезком. Да при склеивании пар второй отрезок в последней паре запросто может быть короче первого *(*и всех остальных*)*, но это ничего не меняет.

SortableList:

- Написать нерекурсивный вариант сортировки двоичным слиянием для списка. Сам алгоритм описан несколькими строчками выше. Конечно, для слияния списков можно (и нужно) воспользоваться уже написанной функцией `merge`.

- В сортировке двоичным слиянием `MergeSort` реализовать вышеописанный вариант разделения списка на две (почти) равные части. Вот этот: в первый список включаем первый, третий, пятый и т.д. элементы, а во второй, соответственно, второй, четвёртый, шестой и т.д. Для этого достаточно связать каждый узел не со следующим элементом, а с запоследующим.

- Хорошо бы реализовать вариант Сэджвика для списка. Суть его вкратце такова: когда длина сортируемого списка меньше некоторого определённого предела (назовём его cutoff), мы не сортируем его, а оставляем, как есть. В результате каждый элемент оказывается удалён от места своей “правильной” дислокации не более, чем на cutoff позиций. В конце сортируем получившийся ряд методом простых вставок. Он работает быстро, поскольку передвигать элемент потребуется совсем недалеко. Проблема тут, конечно, в простых вставках, мы же можем двигаться по списку только вперёд, и, если добавлять элементы от первого элемента списка к последнему, то при поиске места придётся бежать довольно далеко и долго, т.е. выигрыша от предварительной “почтисортировки” QuickSort’ом не получаем, даже наоборот, сильно проигрываем в скорости - за каждым элементом придётся бежать довольно долго. Выход напрашивается сам собюой - формировать список с конца. Для этого, конечно, не будем разворачивать список, а просто будем исполнять вставку рекурсивно, как-то примерно так, как описано в этом псевдокоде:

```
func SimpleInsertionSort(l *SortableList) {
  if хвост списка содержит более одного элемента {
      SimpleInsertionSort (хвост списка *SortableList)
      Вставить начальный элемент списка l* на своё место, пробегая по хвосту э
того списка
  }
}
```

здесь хвост списка - это весь список, кроме первого элемента: хвост списка list = (*list.First).Next.

Задание не очень тривиальное, возникает немало тонких моментов, требует, как обычно, точности, но довольно интересное. Хорошо бы его сделать.

Фактически это даже два задания: модифицировать QuickSort и написать сортировку простыми вставками для однонаправленного списка.